Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de la Universidad

Tarea 1

Integrantes:
Matias Azocar
Bruno Hernandez
Evelyn Lorca
Profesor: Jorge Amaya

Fecha de entrega: 27 de mayo de 2020
Santiago, Chile

Resumen

El siguiente informe presenta el modelamiento, desarrollo y resultados obtenidos de la realización númerica de un problema de optimización dado. Dicho problema corresponde a la Tarea 1 del ramo MA5701 "Optimización no Lineal".

Durante el modelamiento del problema se hizo uso de notaciones y conceptos sobre la teoría de grafos como herramienta para generar una estructura amigable en el contexto. Luego se analizaron las restricciones encontradas y se argumentó del porqué este problema debiese tener solución. Al momento de programar un algoritmo que resolviera el problema fue necesario replantear problemas equivalentes con el fin de aumentar la simpleza de las restricciones y así poder escribir un código más eficiente. El programa se construyó mediante el módulo SciPy de Python, apoyado con la librería Numpy y su amplia gama de operaciones y estructuras.

En la parte del análisis de las soluciones encontradas se discute sobre la preferencia que optó el flujo hacia caminos acotados en vez de arcos irrestrictos y comparar soluciones para dos casos distintos de *costos administrativos*.

Finalmente se entregan los códigos en la Sección de Anexos para ser evaluados y replicados a gusto. En ellos se encentra la aplicación de la librería `networkx` para la creación y gráfica de estructuras de grafos.

1. Problema

Una planta de producción metalúrgica tiene que maximizar el beneficio total del flujo diario de material que enviará desde el nodo 1 hasta el nodo 11, según el grafo de la **Figura** 1. Cada nodo representa una unidad de proceso donde el material es transformado. En cada arco se indica la cota máxima de flujo que puede pasar por el arco correspondiente. Los arcos sin información no tiene cota superior de capacidad, es decir, tienen capacidad infinita. Sin embargo, esos arcos que no tienen cota de capacidad tienen costo de transporte igual a 2 US$ por unidad transportada en el arco.

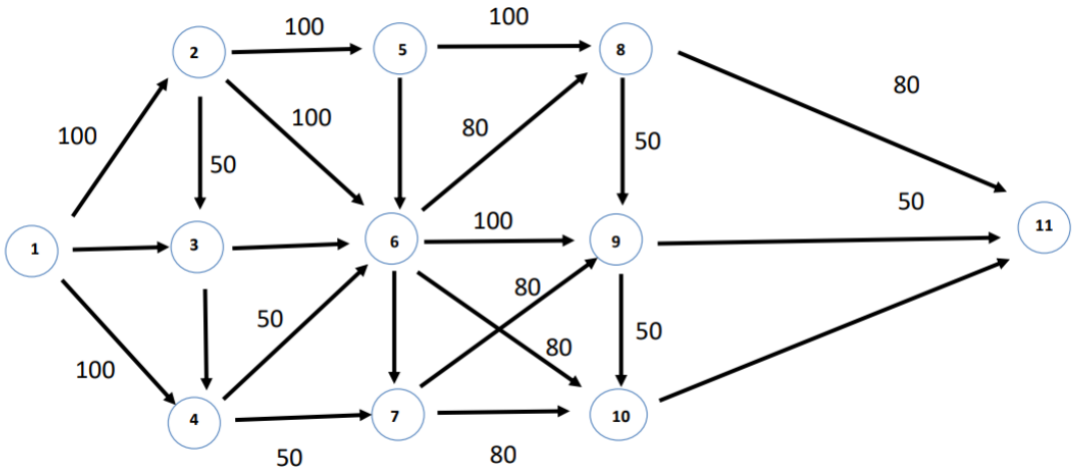


Figura 1: Grafo representate del proceso.

Por razones operacionales, los nodos 6 y 9 no pueden trabajar a la vez (se excluyen mutuamente), es decir, si ese día el nodo 6 procesa flujo, entonces el nodo 9 no lo debe hacer y viceversa.

El costo administrativo de entrega del producto es $k(x) = 20x$ US$, donde x es la cantidad total del producto final entregado. Los beneficios por venta son dependientes de la misma cantidad entregada, según la función:

$$b(x) = 150x - \frac{x^2}{2}$$

2. Modelamiento

Definamos como $N = \{1, \cdots, 11\}$ el conjunto de nodos y consideremos las variables reales contínuas x_{ij} que señala el flujo desde el nodo i al j.
Además consideremos el conjunto $\mathcal{F} = \{(i,j) \in N^2 : \text{ existe camino ente } i \text{ y } j \text{ de un paso}\}$, este conjunto restringe las variables a aquellas que tienen sentido en el contexto del problema. Por ejemplo, podemos notar que no existe flujo que pase desde el nodo 2 al nodo 4 directamente (ver Figura 1) puesto que al material que sale del nodo 2, debe pasar por el nodo 3 o el nodo 6 necesariamente. Del mismo modo, vemos que no hay flujo desde el nodo 6 al nodo 5, pero si en el sentido contrario. En función de esto diremos que si $(i,j) \notin \mathcal{F}$, entonces $x_{ij} = 0$.

Para cada nodo $n \in N$, definamos los conjntos $\mathcal{V}_+(n) = \{i \in N : (i,n) \in \mathcal{F}\}$ y $\mathcal{V}_-(n) = \{i \in N : (n,i) \in \mathcal{F}\}$. Es decir, en $\mathcal{V}_+(n)$ están todos los vecinos que mandan flujo hacia en nodo n y en $\mathcal{V}_-(n)$ todos aquellos que pueden recibir flujo de n.

Si $(i,j) \in \mathcal{F}$, definamos como w_{ij} a la capacidad máxima de material que se puede mandar desde el nodo i hasta el nodo j. Definimos también el costo por unidad de material transportada como

$$c_{i,j} = \begin{cases} 0 & \text{si } w_{ij} < \infty \\ 2 & \text{si } w_{ij} = \infty \end{cases}$$

Notemos que la cantidad total de producto entregado es aquella que llega al nodo 11, entonces decimos que la cantidad total del producto es $\mathbf{x} = \sum_{(i,11)\in\mathcal{V}_+(11)} x_{i,11}$. Además, nos aseguraremos de que todo lo que entre a un nodo termine el recorrido hasta llegar al nodo final. Por último, como restringimos nuestro conjunto de flunjos a los indexados por $\mathcal{F}$, no existirán flujos negativos que se "devuelvan" por algún camino.

Juntando todo esto, definimos el problema de optimización como:

$$\begin{aligned} \underset{(x_{ij})_{i,j}}{Maximizar} \quad & b(\mathbf{x}) - k(\mathbf{x}) - \textstyle\sum_{(i,j)\in\mathcal{F}} c_{ij}\, x_{ij} \\ \text{sujeto a} \quad & x_{ij} \le w_{ij}, && (i,j) \in \mathcal{F}, \\ & \textstyle\sum_{i\in\mathcal{V}_+(k)} x_{ik} = \sum_{j\in\mathcal{V}_-(k)} x_{kj}, && \forall\, k \in N\backslash\{1,11\}, \\ & x_{i,j} \ge 0, && \forall\, (i,j) \in \mathcal{F} \\ & x_{i,j} = 0, && \forall\, (i,j) \in N^2\backslash\mathcal{F}. \end{aligned} \tag{P1}$$

Sin embargo, al Modelo P1, le hace falta una restricción; la interdependencia de los nodos 6 y 9. Dada la segunda restricción de P1, nos basta que el flujo de entrada sea 0 para que un nodo esté *inactivo*. Por lo tanto la restricción, que llamaremos R, la podemos escribir como:

$$\begin{cases} \text{Si } x_{i6} \neq 0 \quad \text{para algún valor } i \in \mathcal{V}_+(6), \quad \text{entonces } x_{j9} = 0 \ \ \forall\, j \in \mathcal{V}_+(9). \\ \\ \text{Si } x_{i9} \neq 0 \quad \text{para algún valor } i \in \mathcal{V}_+(9), \quad \text{entonces } x_{j6} = 0 \ \ \forall\, j \in \mathcal{V}_+(6). \end{cases} \tag{R}$$

Por lo tanto el modelo completo para resolver el problema es (P1) + (R).

3. Resolución del problema

Lo primero a notar en el problema es que la función objetivo definida en el Problema P1 es una función cóncava puesto que es un polinomio de grado 2 en la función b. Por lo tanto existe un único valor máximo.

Además notemos que el conjunto factible es compacto y no vacío. Primero notamos que es no vacío porque el vector nulo, es decir $x_{ij} = 0 \ \forall (i,j) \in N^2$, es factible al cumplir con todas las restricciones. Además notemos que, cuando w_{ij} es finito, x_{ij} es acotado. En otro caso, como cada nodo en $\mathcal{V}_+(11)$ posee capacidad acotada, la segunda restricción nos dice que el flujo en cada arco está acotado, de la siguiente manera:

$$x_{ij} \leq \sum_{k \in \mathcal{V}_+(11)} w_{k,11}, \quad \forall (i,j) \in \mathcal{F}.$$

Por lo tanto, el conjunto de restricciones puede verse como un conjunto compacto en $\mathbb{R}^{N^2}$. Todo lo antes dicho, sumado a la continuidad de la función objetivo, podemos afirmar que el problema posee solución y este valor es único.

3.1. Resolución numérica

La librería `numpy` del lenguaje de programación `Python` nos entrega la estructura numérica con las operaciones algebráicas básicas, estructura de listas y valores infinitos (comando `numpy.inf`).[1]

El primer paso fue definir los parámetros del problema, las matrices W y C, correspondientes a las capacidades máximas y los costos asociados, respectivamente. Ambas pertenecientes al conjunto de matrices cuadradas de dimensión $N \times N$. Esto se logra con el comando `np.zeros([N,N])`[2] (generador de matriz de puros ceros en sus componentes). Notamos que las definiciones de W y C están hechas para el conjunto $\mathcal{F}$, sin embargo estas pueden ser extendidas a N^2 reemplazando los valores faltantes por cero. Podemos definir rápidamente las funciones b y k con el comando `lambda`:

```
b = lambda x: 150*x - (x**2)/2
k = lambda x: 20*x
```

Podemos guardar, para cada valor de N, los nodos que pertenezcan a $\mathcal{V}_+$ y a $\mathcal{V}_-$ según sea el caso. De esta forma es más fácil manipular los índices al momento de generar las funciones de restricción.

Definimos la función objetivo como `objetivo(X)`, que llama a las funciones `b` y `k` antes definidas, y utiliza un ciclo `for` para generar el valor de los costos asociados a los flujos definidos por la matriz `X`. Retornando el valor `-(b - k - costos)`, es decir el inverso aditivo de la función objetivo definida en P1. La explicación de esto será explicada a continuación.

3.1.1. Módulo optimizador `scipy.optimize`

La librería `scipy.optimize` posee la función `minimize`, que recibe una función objetivo, un punto inicial, un vector de cotas y un vector de restricciones y retorna el valor del mínimo de la

[1] Desde ahora y en adelante se utilizará la abreviatura `np` para referirse a la librería `numpy`, así el comando `numpy.inf` ahora será `np.inf`.

[2] Para efectos de código `N = 11`, en cualquier otro contexto N tiene su definición anterior, $N = \{1, \ldots, 11\}$.

función objetivo, una aproximación del punto que optimiza la función y parámetros extras como la aproximación de la matriz jacobiana, el número de iteraciones necesarias, etc.

Dado que la librería sólo nos permite minimizar funciones, hemos decidido cambiar la función objetivo en P1 para replantear el problema como un problema de minimización, cambiando la función por su inverso aditivo y manteniendo las restricciones.

Otro inconveniente de la función `minimize` es que sólo admite funciones cuyo *input* sean vectores, es decir $f : \mathbb{R}^n \to \mathbb{R}$ para algún valor $n \in \mathbb{N}$. Dado que nuestra función `objetivo` y los parámetros están en $\mathcal{M}_{N\times N}(\mathbb{R})$ la función no nos permite aplicar el algoritmo minimizador. Afortunadamente, la librería `numpy` nos puede resolver este problema con la función `reshape` que cambia las dimensiones de las estructuras siempre y cuando la cantidad de elementos de entrada sea al mismo, en nuestro caso 11^2. Además este proceso es invertible, es decir, si vuelvo a ocupar `reshape` a un vector que originalmente era una matriz, puedo recuperar la misma matriz sin riesgo de intercambiar algún elemento de esta.

Con esto en mente, la aplicación de la función `minimize` procede luego de cambiar los parámetros y el *input* de la función `objetivo` para que ahora sea un vector de largo 11^2.

3.1.2. Definición de las restricciones

Como la función `minimize` nos pide vectores con la información de las cotas y las restricciones. Las cotas ya están definidas gracias a nuestra matriz W, además podemos definir una matriz de valores ceros para las cotas inferiores.

Para las restricciones de flujo, creamos un ciclo `for` desde los valores 2 hasta 10, para definir la suma sobre los flujos de entrada y salida de cada nodo. Este proceso fue bastante sencillo dado que ya poseíamos las *listas* que contenían a $\mathcal{V}_+$ y a $\mathcal{V}_-$, sobre las cuales indexamos las sumas correspondientes.

El mayor problema se presenta al escribir la restricción R, dado que sólo un valor de flujo restringe a todos los valores de flujo sobre otro nodo, la cantidad de restricciones asociadas es demasiadas para poder escribirlas de manera eficiente. Es por esto que proponemos una reformulación de esta restricción, como las variables de flujo son positivas, notamos que

$$x_{i6} \neq 0 \text{ para algún valor } i \in \mathcal{V}_+(6) \iff \sum_{i\in\mathcal{V}_+(6)} x_{i6} \neq 0,$$

y además que

$$x_{i9} = 0 \ \forall\, i \in \mathcal{V}_+(9) \iff \sum_{i\in\mathcal{V}_+(9)} x_{i9} = 0.$$

De esta forma, reformulamos la restricción R y la escribimos como:

$$\begin{cases} \text{Si } \sum_{i\in\mathcal{V}_+(6)} x_{i6} \neq 0 \implies \sum_{i\in\mathcal{V}_+(9)} x_{i9} = 0 \\ \\ \text{Si } \sum_{i\in\mathcal{V}_+(9)} x_{i9} \neq 0 \implies \sum_{i\in\mathcal{V}_+(6)} x_{i6} = 0 \end{cases} \tag{R2}$$

La restricción R2 es fácilmente programable por dos instancias `if`. De esta forma el programa está listo para ser ejecutado.

4. Resultados

Antes de mostrar los resultados del proceso de optimización, es necesario aclarar que los algoritmos de descenso ocupados por `Python` aproximan los valores y entrega cifras significativas excesivas en algunos casos, es por esto que asumimos que aquellos valores que acerquen al orden de 10^{-10} serán asumidos como cero, este residuo será utilizado como argumento para la aproximación de las otras variables además.

4.1. Caso $k(x) = 20x$

A continuación entregamos una tabla informativa que muestra aquellos valores de flujo que resultaron no nulos, para el problema 1, con función de entrega de producto $k(x) = 20x$:

Tabla 1: Valores de flujo $k(x) = 20x$

Flujos	Valores
x_{12}	89
x_{14}	41
x_{25}	89
x_{47}	41
x_{58}	89
x_{79}	41
x_{89}	9
$x_{8,11}$	80
$x_{9,11}$	50

La Figura 2 muestra la solución encontrada de manera gráfica, en comparación a la Figura 1, este grafo sólo contiene aquellas aristas por donde efectivamente pasó flujo. Podemos notar que el algoritmo evadió totalmente aquellas aristas que tenían capacidad infinita, esto dado a que el costo aumenta cada vez que una de ellas es usada.

Además notamos que el nodo utilizado fue el 9 y no el nodo 6, una posible explicación de esta decisión puede ser el directa conección que posee el nodo 9 con el nodo final 11. En caso de utilizar el nodo 6, obliga al algoritmo a mandar flujo desde el nodo 10 hacia el nodo 11, cuya arista tiene capacidad infinita y, por ende, costo por su uso, lo que empeoraría la función objetivo.

Por último notamos que el flujo total ocupa en su totalidad la capacidad de los arcos 8-11 y 9-11, resultado que es posible dado que los arcos iniciales (aquellos que inciden en 1) suman mayor capacidad que esta (notar que la segunda restricción de P1 implica que estos deben ser iguales).

El valor de la función objetivo en este caso es de US$8450.

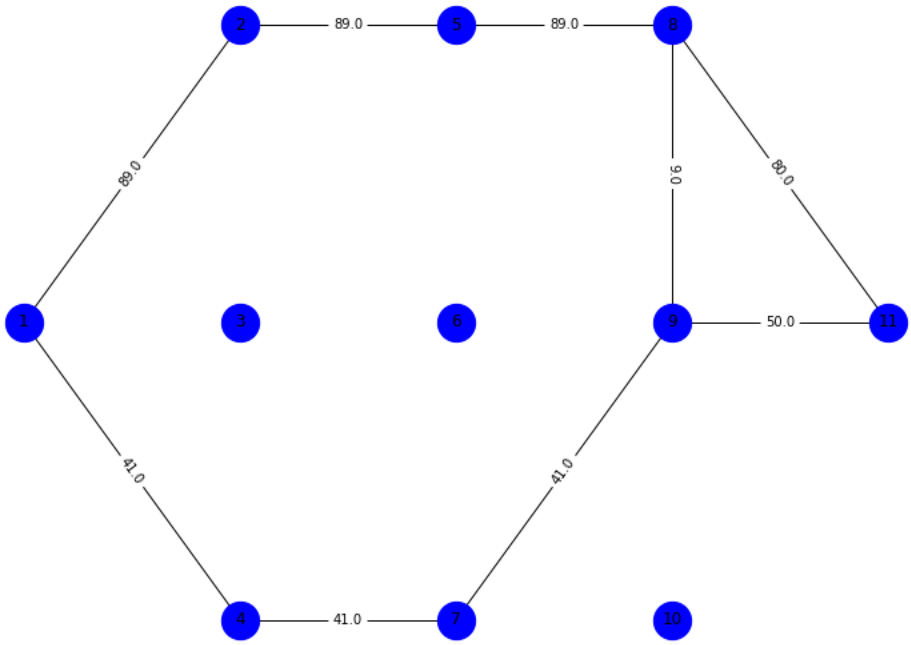


Figura 2: Solución gráfica 1.

4.2. Caso $k(x) = 40x$

El siguiente caso muestra un cambio en la función de costo administrativo, específicamente; éste se duplica. La siguiente tabla muestra el flujo óptimo encontrado por el algoritmo para el caso $k(x) = 40x$:

Tabla 2: Valores de flujo $k(x) = 40x$

Flujos	Valores
x_{12}	62
x_{14}	48
x_{25}	62
x_{47}	48
x_{58}	62
x_{79}	48
x_{89}	1
$x_{8,11}$	61
$x_{9,11}$	49

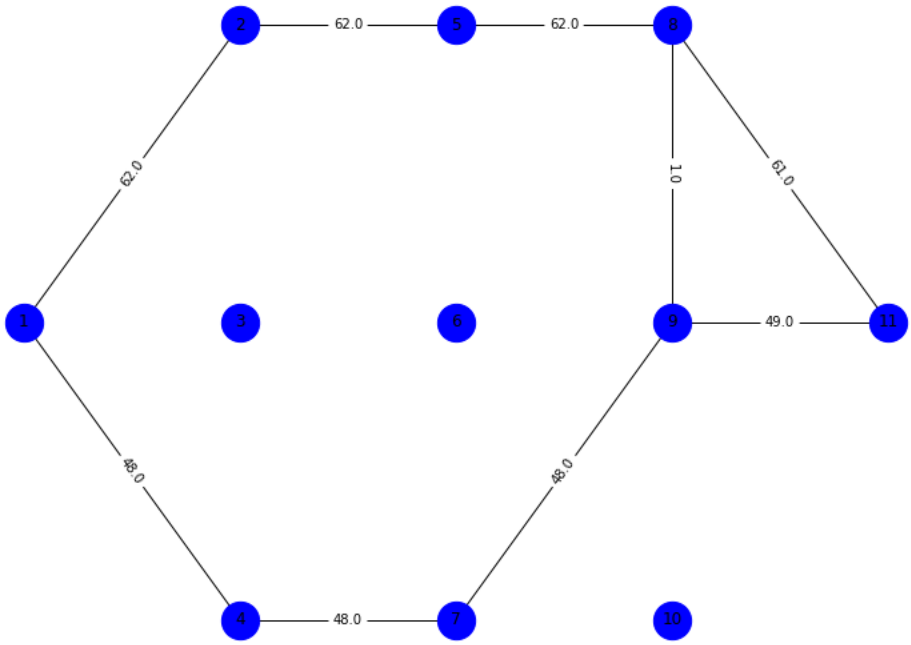


Figura 3: Solución gráfica 2.

De la Figura 3 notamos que la gran diferencia entre el flujo anterior y este es el flujo total que llega al nodo 11. Como era de esperarse, al aumentar el costo administrativo por cantidad de material, la tendencia que debería tomar el flujo total es a bajar para contrarestar este aumento. En cuanto al camino que se utiliza para llegar al proceso final, éste parece el óptimo al momento de evitar los arcos con costo por uso. Evidentemente el valor de la función objetivo diminuye, por el aumento del costo y la disminución del flujo total, llegado a un óptimo de US$6050.

4.3. Comparación de resultados

Dado que ambos casos encuentran su flujo óptimo bajo los mismos caminos, podemos comparar el flujo, arco por acro, y el valor final para la función objetivo.

Tabla 3: Comparación de casos

Flujos	Valores G1	Valores G2
x_{12}	89	62
x_{14}	41	48
x_{25}	89	62
x_{47}	41	48
x_{58}	89	62
x_{79}	41	48
x_{89}	9	1
$x_{8,11}$	80	61
$x_{9,11}$	50	49
Valor Objetivo	8450	6050

5. Anexos

A continuación se incluyen los códigos utilizados.

0.1 Caso 1: $k(x) = 20x$

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

N=11
W = np.zeros([N,N]) #Pesos de los arcos
W[0,1] = 100
W[0,2] = np.inf
W[0,3] = 100
W[1,2] = 50
W[1,4] = 100
W[1,5] = 100
W[2,3] = np.inf
W[2,5] = np.inf
W[3,5] = 50
W[3,6] = 60
W[4,5] = np.inf
W[4,7] = 100
W[5,7] = 80
W[5,8] = 100
W[5,6] = np.inf
W[6,8] = 80
W[6,9] = 80
W[7,8] = 50
W[7,10] = 80
W[8,10] = 50
W[8,9] = 50
W[9,10] = np.inf

k = lambda x: 20*x
b = lambda x: 150*x - (x**2)/2

c = np.zeros([N,N]) #Definición de los costos
for i in range(N):
    for j in range(N):
        if W[i,j] == np.inf:
            c[i,j] = 2

Vint = np.zeros(N,dtype = list) #Corte de entrada para cada nodo
Vout = np.zeros(N,dtype = list) #Corte de salida para cada nodo

for i in range(N):
    vin = []
    vout = []
    for j in range(N):
```

```
        if W[j,i] != 0:
            vin.append(j)
        if W[i,j] != 0:
            vout.append(j)
    Vint[i] = vin
    Vout[i] = vout

def objetivo(V):
    '''
    Inicialmente era para una matrix de NxN pero para ser utilizado en el␣
 ↪algoritmo de optimizacion
    modificamos esta matriz a un vector de N^2, luego recuperamos la matriz␣
 ↪dentro de la función
    '''
    X = V.reshape([N,N])
    x = np.sum([X[i,10] for i in Vint[10]])
    costo = np.sum([c[i,j]*X[i,j] for j in range(N) for i in range(N)])
    return -(b(x) - k(x) - costo)

##### Modelo scipy

## Cotas:
lb = np.zeros([N,N]).reshape(N*N)
ub = W.reshape(N*N)
B = list(zip(lb,ub))

## Restricciones:
cons = []
for i in range(1,N-1):
    def f(V, i=i):
        X = V.reshape([N,N])
        return np.sum([X[k,i] for k in Vint[i]]) - np.sum([X[i,k] for k in␣
 ↪Vout[i]])
    cons.append( {'type':'eq', 'fun':f})

def g(V):
    X = V.reshape([N,N])
    if np.sum([X[i,8] for i in Vint[8]]) != 0:
        return np.sum([X[i,5] for i in Vint[5]])
    elif np.sum([X[i,5] for i in Vint[5]]) != 0:
        return np.sum([X[i,8] for i in Vint[8]])
    else:
        return X[10,10]

cons.append( {'type':'eq', 'fun':g})

minimo = minimize(objetivo, np.zeros(N*N), bounds = B , constraints = cons)
```

```
xmin = minimo.x.reshape([N,N])
for i in range(N):
    for j in range(N):
        if round(xmin[i,j]) != 0:
            print('Variable x_'+str(i+1)+'-'+str(j+1)+' = ', round(xmin[i,j]))


print('Valor objetivo: ', -1*round(minimo.fun))
```

```
Variable x_1-2 =  89.0
Variable x_1-4 =  41.0
Variable x_2-5 =  89.0
Variable x_4-7 =  41.0
Variable x_5-8 =  89.0
Variable x_7-9 =  41.0
Variable x_8-9 =  9.0
Variable x_8-11 =  80.0
Variable x_9-11 =  50.0
Valor objetivo:  8450
```

0.2 Gráfico del flujo

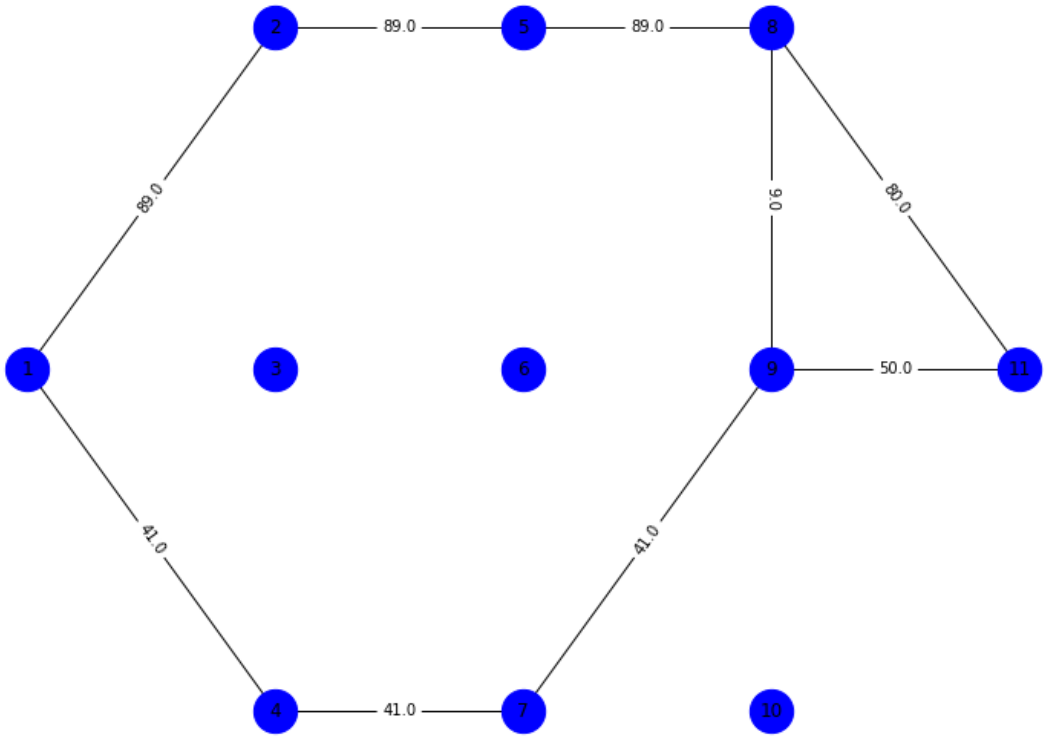
[2]:
```
import networkx as nx
G = nx.Graph()
G.add_node(1, pos = (0,1))
G.add_node(2, pos = (1,2))
G.add_node(3, pos = (1,1))
G.add_node(4, pos = (1,0))
G.add_node(5, pos = (2,2))
G.add_node(6, pos = (2,1))
G.add_node(7, pos = (2,0))
G.add_node(8, pos = (3,2))
G.add_node(9, pos = (3,1))
G.add_node(10, pos = (3,0))
G.add_node(11, pos = (4,1))

for i in range(N):
    for j in range(N):
        if W[i,j] != 0.:
            if round(xmin[i,j]) != 0:
                G.add_edge(i+1,j+1, weight= round(xmin[i,j]))
#             else:
#                 G.add_edge(i+1,j+1, weight= round(xmin[i,j]))
```

[4]:
```
plt.figure(figsize = (10,7))
pos=nx.get_node_attributes(G,'pos')
```

```
nx.draw(G,pos, node_color = 'b',with_labels=True, node_size = 800)
labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)
```

[4]:
```
{(1, 2): Text(0.5, 1.5, '89.0'),
 (1, 4): Text(0.5, 0.5, '41.0'),
 (2, 5): Text(1.5, 2.0, '89.0'),
 (4, 7): Text(1.5, 0.0, '41.0'),
 (5, 8): Text(2.5, 2.0, '89.0'),
 (7, 9): Text(2.5, 0.5, '41.0'),
 (8, 9): Text(3.0, 1.5, '9.0'),
 (8, 11): Text(3.5, 1.5, '80.0'),
 (9, 11): Text(3.5, 1.0, '50.0')}
```



0.3 Caso 2: $k(x) = 40x$

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

N=11
```

```
W = np.zeros([N,N]) #Pesos de los arcos
W[0,1] = 100
W[0,2] = np.inf
W[0,3] = 100
W[1,2] = 50
W[1,4] = 100
W[1,5] = 100
W[2,3] = np.inf
W[2,5] = np.inf
W[3,5] = 50
W[3,6] = 60
W[4,5] = np.inf
W[4,7] = 100
W[5,7] = 80
W[5,8] = 100
W[5,6] = np.inf
W[6,8] = 80
W[6,9] = 80
W[7,8] = 50
W[7,10] = 80
W[8,10] = 50
W[8,9] = 50
W[9,10] = np.inf

k = lambda x: 40*x
b = lambda x: 150*x - (x**2)/2

c = np.zeros([N,N]) #Definición de los costos
for i in range(N):
    for j in range(N):
        if W[i,j] == np.inf:
            c[i,j] = 2

Vint = np.zeros(N,dtype = list) #Corte de entrada para cada nodo
Vout = np.zeros(N,dtype = list) #Corte de salida para cada nodo

for i in range(N):
    vin = []
    vout = []
    for j in range(N):
        if W[j,i] != 0:
            vin.append(j)
        if W[i,j] != 0:
            vout.append(j)
    Vint[i] = vin
    Vout[i] = vout
```

```
def objetivo(V):
    '''
    Inicialmente era para una matrix de NxN pero para ser utilizado en el␣
 ↪algoritmo de optimizacion
    modificamos esta matriz a un vector de N^2, luego recuperamos la matriz␣
 ↪dentro de la función
    '''
    X = V.reshape([N,N])
    x = np.sum([X[i,10] for i in Vint[10]])
    costo = np.sum([c[i,j]*X[i,j] for j in range(N) for i in range(N)])
    return -(b(x) - k(x) - costo)

##### Modelo scipy

## Cotas:
lb = np.zeros([N,N]).reshape(N*N)
ub = W.reshape(N*N)
B = list(zip(lb,ub))

## Restricciones:
cons = []
for i in range(1,N-1):
    def f(V, i=i):
        X = V.reshape([N,N])
        return np.sum([X[k,i] for k in Vint[i]]) - np.sum([X[i,k] for k in␣
 ↪Vout[i]])
    cons.append( {'type':'eq', 'fun':f})

def g(V):
    X = V.reshape([N,N])
    if np.sum([X[i,8] for i in Vint[8]]) != 0:
        return np.sum([X[i,5] for i in Vint[5]])
    elif np.sum([X[i,5] for i in Vint[5]]) != 0:
        return np.sum([X[i,8] for i in Vint[8]])
    else:
        return X[10,10]

cons.append( {'type':'eq', 'fun':g})

minimo = minimize(objetivo, np.zeros(N*N), bounds = B , constraints = cons)

xmin = minimo.x.reshape([N,N])
for i in range(N):
    for j in range(N):
        if round(xmin[i,j]) != 0:
            print('Variable x_'+str(i+1)+'-'+str(j+1)+' = ', round(xmin[i,j]))
```

```
print('Valor objetivo: ', -1*round(minimo.fun))
```

```
Variable x_1-2 =  62.0
Variable x_1-4 =  48.0
Variable x_2-5 =  62.0
Variable x_4-7 =  48.0
Variable x_5-8 =  62.0
Variable x_7-9 =  48.0
Variable x_8-9 =  1.0
Variable x_8-11 =  61.0
Variable x_9-11 =  49.0
Valor objetivo:  6050
```

0.4 Gráfico del flujo

[6]:
```
G = nx.Graph()
G.add_node(1, pos = (0,1))
G.add_node(2, pos = (1,2))
G.add_node(3, pos = (1,1))
G.add_node(4, pos = (1,0))
G.add_node(5, pos = (2,2))
G.add_node(6, pos = (2,1))
G.add_node(7, pos = (2,0))
G.add_node(8, pos = (3,2))
G.add_node(9, pos = (3,1))
G.add_node(10, pos = (3,0))
G.add_node(11, pos = (4,1))

for i in range(N):
    for j in range(N):
        if W[i,j] != 0.:
            if round(xmin[i,j]) != 0:
                G.add_edge(i+1,j+1, weight= round(xmin[i,j]))
```

[7]:
```
plt.figure(figsize = (10,7))
pos=nx.get_node_attributes(G,'pos')
nx.draw(G,pos, node_color = 'b',with_labels=True, node_size = 800)
labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)
```

[7]:
```
{(1, 2): Text(0.5, 1.5, '62.0'),
 (1, 4): Text(0.5, 0.5, '48.0'),
 (2, 5): Text(1.5, 2.0, '62.0'),
 (4, 7): Text(1.5, 0.0, '48.0'),
 (5, 8): Text(2.5, 2.0, '62.0'),
 (7, 9): Text(2.5, 0.5, '48.0'),
```

```
(8, 9): Text(3.0, 1.5, '1.0'),
(8, 11): Text(3.5, 1.5, '61.0'),
(9, 11): Text(3.5, 1.0, '49.0')}
```